

ADANA SCIENCE AND TECHNOLOGY UNIVERSITY

Introduction to Computer Programming II

Objectives for today

- Pointers
  - Declaration
  - Reference and Dereferene operators
- Pointers and arrays
- Pointer arithmetics

POINTERS : Introduction

- Variables have been explained as locations in the computer's memory which can be accessed by their identifier (their name).
- This way, the program does not need to care about the physical address of the data in memory;
- it simply uses the identifier whenever it needs to refer to the variable.

POINTERS : Introduction

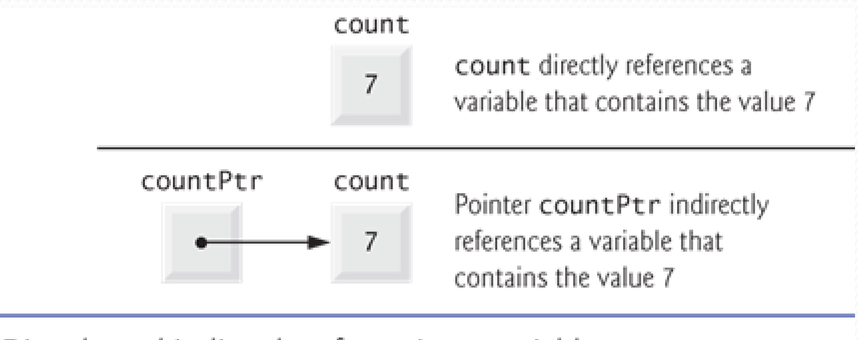
- For a C++ program, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address.
- These single-byte memory cells are ordered in a way that allows data representations larger than one byte to occupy memory cells that have consecutive addresses.
- When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address).
- Generally, C++ programs do not actively decide the exact memory addresses where its variables are stored.

POINTERS : Introduction

- Fortunately, that task is left to the environment where the program is run –
  - generally, an operating system that decides the particular memory locations on runtime.
  - However, it may be useful for a program to be able to obtain the address of a variable during runtime in order to access data cells that are at a certain position relative to it.

Pointer Variable Declarations and Initialization

- A pointer contains the memory address of a variable that, in turn, contains a specific value.
- In this sense, a variable name **directly** references a value, and a pointer **indirectly** references a value.
- Referencing a value through a pointer is called indirection.
- Diagrams typically represent a pointer as an arrow from the variable that contains an address to the variable located at that address in memory.



Directly and indirectly referencing a variable.

Pointer Variable Declarations and Initialization (cont.)

- The declaration
  - `int *countPtr, count;`

  declares the variable `countPtr` to be of type `int *` (i.e., a pointer to an `int` value) and is read as "`countPtr` is a pointer to `int`."
  - Variable `count` in the preceding declaration is declared to be an `int`, not a pointer to an `int`.
  - The `*` in the declaration applies only to `countPtr`.
  - Each variable being declared as a pointer must be preceded by an asterisk (`*`).
- When `*` appears <u>in a declaration</u>, <u>it isn't an operator</u>; rather, it indicates that the variable being declared is a pointer.
- Pointers can be declared to point to objects of any data type.

Pointer Variable Declarations and Initialization (cont.)

- Pointers should be initialized either when they're declared or in an assignment.
- A pointer may be initialized to `0`, `NULL` or an address of the corresponding type.
- A pointer with the value `0` or `NULL` points to nothing and is known as a null pointer.
  - `NULL` is equivalent to `0`, but in C++, `0` is used by convention.
- The value `0` is the only integer value that can be assigned directly to a pointer variable without first casting the integer to a pointer type.

Pointer Operators

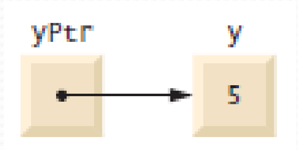
- The address operator (&) is a unary operator that obtains the memory address of its operand.
- Assuming the declarations

```
int y = 5; // declare variable y
int *yPtr; // declare pointer variable yPtr
```

  the statement

```
yPtr = &y; // assign address of y to yPtr
```

  assigns the address of the variable `y` to pointer variable `yPtr`.



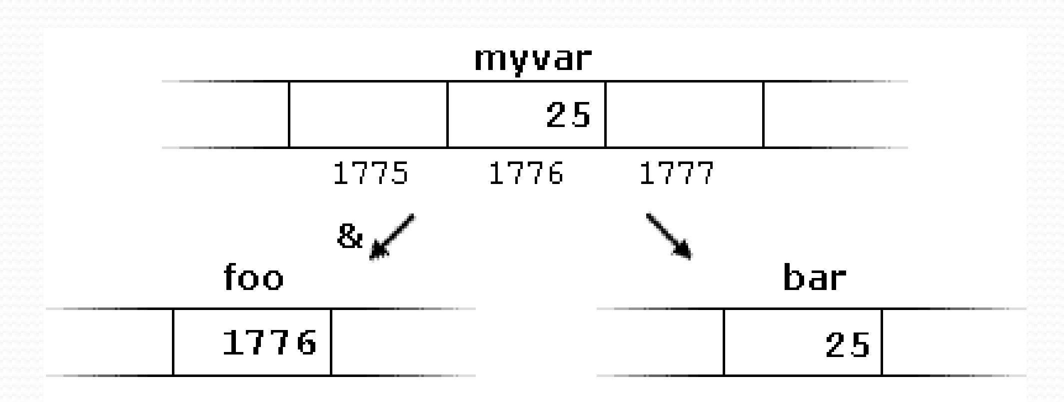
- shows a schematic representation of memory after the preceding assignment.

Address-of operator (&)

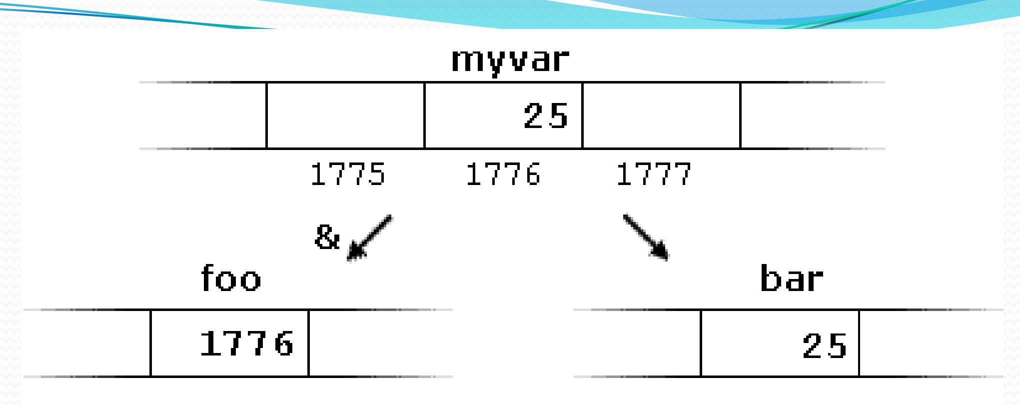
- The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as *address-of operator*.
- For example:
  - foo = &myvar;
- This would assign the address of variable myvar to foo;
  - by preceding the name of the variable myvar with the *address-of operator* (&),
  - we are no longer assigning the content of the variable itself to foo, but its address.

Address-of operator (&)

- The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that *myvar* is placed during runtime in the memory address 1776.
- In this case, consider the following code fragment:
  - `myvar = 25;`
  - `foo = &myvar;`
  - `bar = myvar;`



```
myvar = 25;
foo = &myvar;
bar = myvar;
```



- First, we have assigned the value 25 to ***myvar*** (*a variable whose address in memory we assumed to be 1776*).
- The second statement assigns ***foo*** the address of ***myvar***, which we have assumed to be 1776.
- Finally, the third statement, assigns the value contained in ***myvar*** to ***bar***. This is a standard assignment operation, as already done many times earlier.
- The main difference between the second and third statements is the appearance of the *address-of operator* (&).

Pointer Operators (cont.)

location 500000	yPtr: 600000	location 600000	y: 5

Representation of y and yPtr in memory.

- Figure shows another pointer representation in memory with integer variable `y` stored at memory location `600000` and pointer variable `yPtr` stored at memory location `500000`.
- The address operator cannot be applied to constants or to expressions that do not result in references.
- The `*` operator, commonly referred to as the indirection operator or dereferencing operator, returns a synonym for the object to which its pointer operand points.
  - Called dereferencing a pointer
- A dereferenced pointer may also be used on the left side of an assignment.

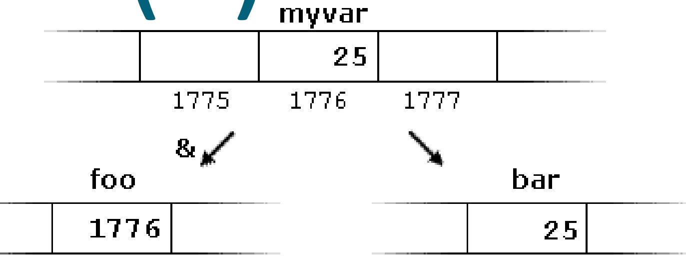
Dereference operator (*)

- Remember
  - a variable which stores the address of another variable is called a *pointer*.
- Pointers are said to "point to" the variable whose address they store.
- An interesting property of pointers is that they can be used to access the variable they point to directly.
- This is done by preceding the pointer name with the *dereference operator* (*).
- The operator itself can be read as "value pointed to by".

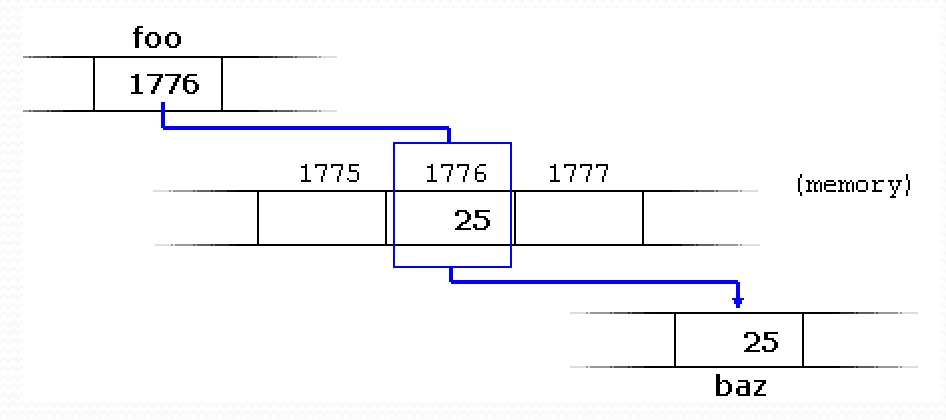
Dereference operator (*)

Remember the previous example

```
myvar = 25;
foo = &myvar;
bar = myvar;
```



- The following statement:
  - baz = *foo;
- could be read as: "*baz* equal to the value pointed to by *foo*",
- and the statement would actually assign the value 25 to *baz*,
  - since foo is 1776,
  - and the value pointed to by 1776 would be 25.



Dereference operator (*)

- It is important to clearly differentiate that ***foo*** refers to the value <u>1776</u>,
- while ****foo*** (with an asterisk * preceding the identifier) <u>refers to the value stored at address 1776</u>
  - which in this case is 25.
- Notice the difference of including or not including the *dereference operator*
  - baz = foo; // baz equal to foo (1776)
  - baz = *foo; // baz equal to value pointed to by foo (25)

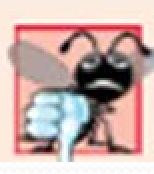
Refence (&) and Dereference (*) Operators

- The reference and dereference operators are thus complementary:
  - & is the *address-of operator*, and can be read simply as "address of"
  - * is the *dereference operator*, and can be read as "value pointed to by"
- Thus, they have sort of opposite meanings: An address obtained with & can be dereferenced with *.



Common Programming Error 8.2

Dereferencing an uninitialized pointer could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion, possibly with incorrect results.



Common Programming Error 8.3

An attempt to dereference a variable that is not a pointer is a compilation error.



Common Programming Error 8.4

Dereferencing a null pointer is often a fatal execution-time error.

```cpp
// Fig. 8.4: fig08_04.cpp
// Pointer operators & and *.
#include <iostream>
using namespace std;

int main()
{
   int a; // a is an integer
   int *aPtr; // aPtr is an int * which is a pointer to an integer

   a = 7; // assigned 7 to a
   aPtr = &a; // assign the address of a to aPtr

   cout << "The address of a is " << &a
      << "\nThe value of aPtr is " << aPtr;
   cout << "\n\nThe value of a is " << a
      << "\nThe value of *aPtr is " << *aPtr;
   cout << "\n\nShowing that * and & are inverses of "
      << "each other.\n&*aPtr = " << &*aPtr
      << "\n*&aPtr = " << *&aPtr << endl;
} // end main
```

Fig. 8.4 | Pointer operators & and *. (Part 1 of 2.)

```
The address of a is 0012F580
The value of aPtr is 0012F580

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

Fig. 8.4 | Pointer operators & and *. (Part 2 of 2.)

Pointer Operators (cont.)

- The & and * operators are inverses of one another.
- The address (&) and dereferencing operator (*) are unary operators on the third level.
- Precedence and associativity of the operators are given by:

Operators	Associativity	Type
() []	left to right	highest
++ -- static_cast< *type* >(*operand*)	left to right	unary (postfix)
++ -- + - ! & *	right to left	unary (prefix)
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	insertion/extraction
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Pointers: Declaration Example

a	b	p
?	?	?
1000	1001	1002

Memory Unit

Memory Address

```
int a, b;
int *p;
p = &a;
*p = 5;
b = *p;
```

Example

```
#include <iostream>
using namespace std;

int main()
{
    int a=25;
    int b = a;
    int *c=&a;
    int *d=&b;
    *d = 45;

    cout<<"&a = "<<&a<<endl;
    cout<<"&b = "<<&b<<endl;
    cout<<"&c = "<<&c<<endl;
    cout<<"&d = "<<&d<<endl;

    return 0;
}
```

```
&a = 0x28ff0c
&b = 0x28ff08
&c = 0x28ff04
&d = 0x28ff00

Process returned 0 (0x0)   execution time : 0.017 s
Press any key to continue.
```

Fill out the table below using the codes and the output

Name of the variable :					
Value of the variable :					
Address of the variable :	0x28ff00	0x28ff04	0x28ff08	0x28ff0c	0x28ff0e

```
int *b;
int num=453;
b=&num;
cout << b  //1005
cout << *b //453
```

Then:

```
cout << *num << endl;
cout << *&num << endl;
cout << &*num << endl;
cout << &num << endl;
```

Pointers : A little Bit More

- Due to the ability of a pointer to directly refer to the value that it points to, a pointer has different properties when it points to a char than when it points to an int or a float.
- Once dereferenced, the type needs to be known.
- And for that, the declaration of a pointer needs to include the data type the pointer is going to point to.
- Remember that, the declaration of pointers follows this syntax:
  - type * name;
    - where type is the data type pointed to by the pointer.
    - This type **is not the type of the pointer itself**, but the type of the data the pointer points to.

Pointers : A little Bit More

- Examples of declarations of pointers.
  - int * number;
  - char * character;
  - double * decimals;
- Each one is intended to point to a different data type, but, in fact, all of them are pointers and all of them are likely **going to occupy the same amount of space in memory**
  - the size in memory of a pointer depends on the platform where the program runs.
- Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type: the first one points to an int, the second one to a char, and the last one to a double.
- Therefore, although these three example variables are all of them pointers, they actually have different types: int*, char*, and double* respectively, depending on the type they point to.

Example

```
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

- Value of the pointer can be changed during the program
  - Variable it points changes
  - In the example *mypointer* points to *firstvalue* first
  - than it points to *secondvalue*.

```
mypointer = &firstvalue;   *mypointer = 10;
mypointer = &secondvalue;   *mypointer = 20;
```

- Notice that even though neither *firstvalue* nor *secondvalue* are directly set any value in the program, both end up with a value set indirectly through the use of *mypointer*.
- This is how it happens:
  - First, *mypointer* is assigned the address of *firstvalue* using the address-of operator (&).
  - Then, the value pointed to by *mypointer* is assigned a value of 10.
  - Because, at this moment, *mypointer* is pointing to the memory location of *firstvalue*, this in fact modifies the value of *firstvalue*.
- In order to demonstrate that a pointer may point to different variables during its lifetime in a program, the example repeats the process with *secondvalue* and that same pointer *mypointer*.

Example

- Here is an example a little bit more elaborated:

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 = value pointed to by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

- Notice that there are expressions with pointers p1 and p2, both with and without the *dereference operator* (*).
- The meaning of an expression using the *dereference operator* (*) is very different from one that does not.
- When this operator precedes the pointer name, the expression refers to the value being pointed, while when a pointer name appears without this operator, it refers to the value of the pointer itself
  - the address of what the pointer is pointing to

Attention

- Attention to the line:
  - int * p1, * p2;
- This declares the two pointers used in the previous example.
- But notice that there is an asterisk (*) for each pointer, in order for both to have type int* (pointer to int).
- This is required due to the precedence rules.
- Note that if, instead, the code was:
  - int * p1, p2;
    - p1 would indeed be of type int*, but p2 would be of type int.
    - Spaces do not matter at all for this purpose.
    - Simply remember to put one asterisk per pointer.


Q&A